1987

Partitioning the Process of Interaction: An Implementation

Balachander Krishnamurthy

Report Number:
87-706

PARTITIONING THE PROCESS OF INTERACTION: AN IMPLEMENTATION

Balachander Krishnamurthy

CSD-TR-706
September 1987

Partitioning the Process of Interaction: An Implementation

Balachander Krishnamurthy

Department of Computer Sciences
Purdue University
W Lafayette, IN 47907

September 7, 1987

Abstract

We present the implementation of a window system constructed on the model of an abstraction mechanism identifying the layers of interaction between the user and application programs in a workstation environment. The first goal of the abstraction mechanism (explained in a companion paper) is reducing the task of the interface programmer and providing a uniform and customizable interface to a variety of application programs. Secondly, the mechanism aids in minimizing interaction related information in the application programs and in reducing the task of the application programmer as well.

Along with the details of the abstraction mechanisms of our model we discuss the prototype window system constructed on the basis of the abstractions. The various interaction styles—input and output—are discussed, both for interaction with the application programs as well as the window system itself.

1 Introduction

This paper explains in detail an implementation of a window system based on the model outlined in [3]. In our model of interaction we sought abstraction mechanisms to identify the communication protocol between the application programs and the users. Our goal was to provide a uniform interface to a variety of application programs and we required the interface to be customizable as well. We also wanted to minimize the interaction related information in the application program and reduce the tasks of the application programmer as well as the interface programmer.

In this paper we describe the design and implementation of UNCLE, a prototype interactive system built on the package configuration file and user configuration file abstractions of our model. Along with a detailed look at the abstractions we present construction of different styles of interaction that user can use in interacting with various application programs.

The reminder of the paper is divided into five sections. We begin with a brief look at the sample application programs that we used in our prototype. This is followed by an in-depth look at the two abstractions—the package configuration file (PCF) and the user configuration file (UCF). The implementation of our window system, named UNCLE, is presented along with a look at the various styles of interaction. After a look at how a user can change his entire environment we conclude with observations on the prototype and the adherence of the implementation to our model.

2 Example application programs

We chose four representative application programs to demonstrate our model. Our choice of these four application programs was not random—the author has used them extensively and is thus familiar with them. One of the four application programs (*Omicron*) was designed and jointly constructed by the author and a colleague. More to the point, the selected application programs are widely used in our environment.

The example application programs do not preclude other application programs from being chosen for the same purpose. The four sample application programs we chose were *MH*, a mail handler, *RCS*, a revision control system, *Omicron*, an event based scheduler, and the *Xinu-shell*, the com-

mand interpreter for version 7 of the XINU operating system. In the rest of this section we take a brief look at the four application programs.

2.1 MH

Mail handlers are one of the most frequently used programs in interactive systems. As a signficant amount of communication is done through electronic mail, the interface to the mail system has become critical.

MH [6] is a message handling system developed at Rand Corporation. In the traditional mode of handling electronic message a user would enter a mail handling environment, execute mail related actions and exit the environment. *MH* permits the user to intersperse other command interpreter commands with *MH* commands. In building a front end on top of *MH* we switched to the notion of having a separate environment for handling mail. We consider *mail* as a separable component that deserves its own context.

MH tries to simulate real life mail handling by providing *folders*, whereby users can keep messages ordered contextwise. *MH* has a notion of current message number as well as a current folder and the user can change both. Several of the *MH* commands take the current message number in the folder as an implicit argument while some commands require an explicit argument. The default argument is the *current* message. Similarly, *MH* commands that deal with folders take the current folder as the default argument. The *MH* commands are shown in Table 1.

2.2 RCS

Our next sample application program is a revision control system. As large sites frequently have a group of programmers working collectively on a piece of software, revision control systems are required to ensure co-operative working. Revision control systems provides users with unique access to files and stores delta information about changes to restore previous versions of files.

The Revision Control System (RCS) [7] manages multiple revisions of text files by automating the storage, retrieval, merging, and identification of revisions of frequently revised text such as documentation, code. The commands in the RCS subsytem are shown in Table 2.

Table 1: MH commands summary

Message commands	Action
show	Displays a message
next	Moves to the next message in the current folder
prev	Moves to the previous message in the current folder
comp	Creates a message template
rmm	Removes a message
repl	Constructs a reply template to a message
forw	Forwards a message
Folder commands	**Action**
refile	Moves messages between folder
folder	Changes the current folder
scan	Displays the list of message headers in current folder

Table 2: RCS commands summary

RCS command	Action
ci	checks in a new revision of a file
co	checks out a revision of a file
ident	identify files
rcs	creates new RCS files or changes attributes of existing file
rcsclean	cleans up working files
rcsdiff	compares RCS revisions
rcsmerge	merges RCS revisions
rlog	prints log messages and other information about RCS files

2.3 Omicron

Omicron is a tool to automate task execution in a system. We have extended the basic notion of an automatic scheduling tool. *Omicron* is designed to give the user flexibility to specify different types and combinations of events. An in depth description of the design and construction of *Omicron* can be found in [5].

Commands in *Omicron* are shown in Table 3.

Table 3: Omicron commands summary

Omicron commands	Action
lsev	lists the set of valid events od the user
addev	permits addition of new events
rmev	removes the specified event(s)
suspev	suspends the specified event(s)
fgev	foregrounds the specified event(s)
readev	reads and updates a list of event specifications

2.4 The Xinu shell

The last application program we consider is a command interpreter. A command interpreter is a front end to the operating system used to interact with all the underlying utilities in the system and other application programs. In a window system we discount the reliance on the command interpreter by providing explicit application program specific interfaces. However, there are reasons to retain the notion of a command interpreter. A more detailed look at command interpreters within the context of an interactive system can be found in [4].

The command interpreter that we consider here as a sample application program is the Xinu shell. The Xinu Shell is the command interpreter for Version 7 of the Xinu operating system [1]. The Xinu operating system is downloaded into small processors like LSI and users interact with it via a shell like interface their terminals. We wanted to test the hypothesis that our model of interaction would be functional in the case of an application

program running on an LSI under a different operating system. Also, the novelty providing an advanced interface with menus without having to add any code to the Xinu system was motivation for choosing the Xinu shell as one of our sample application program.

The commands in the Xinu shell are similar to the ones found in UNIX command interpreters. We divided the commands into six categories. Table 4 shows the categories and examples of commands in each section.

Table 4: XINU shell commands summary

command category	Sample commands
File related	*cat, cp, rm*
Process control	*create, kill, sleep*
Status	*date, who, netstat*
Device and namespace	*close, mount*
Session related	*exit, reboot*
Documentation oriented	*help*

In application programs such as RCS and Xinu, where commands have several options, and arguments, the rigorous definitions are helpful in presenting the user with a consistent interface for the various commands.

3 Package configuration file—PCF

Having looked at the sample application programs of our interactive system, we consider the internals of our model. We begin by describing a key component of our model—the package configuration file (PCF). The PCF gathers the salient components of the interaction process that includes the syntactic aspects of commands, flags, options, arguments, as well as documentation and error messages. It *does not* dictate the interface between the user and the application program. The PCF specifies the protocol between the window system and the individual application program. The interface designer is responsible for the construction of the PCF. Once the PCFs are constructed, a generic parser which is also part of UNCLE is used to parse

the PCFs and download the information about the application programs into the window system.

The motivation behind the PCF has been outlined in [3]. A package configuration file helps partition the task of a window system and the application program. The PCF is our concrete representation of the firewall that separates the application programmer from the interface programmer and the user. The PCF delineates the task of the application programmer and the interface programmer. By specifying the limits of the application programmer we can ensure two things. First, the application programmer does not have to worry about the process of interaction. His task is to write the application program. Secondly, he *cannot* restrict, or otherwise constrain the various interfaces that can be built to interact with his application program. He cannot wire in decisions that may affect other application programs. An example of a low level (wired in) binding frequently found in interactive systems, is one made to the *completion* event, which completes a partially typed string to one of a set of valid strings. We will consider this example in depth later.

The PCF lowers the interface programmer's work in providing the user with different styles of interaction. As the configuration file format is flexible, it is our contention that interface programmer would *want* to use the PCF to specify the interaction components, viz, the commands, flags, options etc. In return he can use the generic parser to do the style generation automatically. Hooks are provided to alter the styles dynamically. The user is free to impose his own style of interaction with the application program. The interaction can be *user-specific*, *statically configurable*, and *dynamically alterable*. Both input and output are customizable. The user can have a uniform style of interaction with all application programs regardless of the original interface of the application program or the original intent of the application programmer.

The PCF helps the interface designer specify in a high level language the set of valid commands and arguments in the package. The PCF includes the generic command that can be used to invoke the underlying function (the name of the function to be executed will also be part of the PCF), a brief explanatory help message, an error message to be generated when the associated argument validation code fails, and documentation explaining the purpose of the command. The PCF will be parsed *a priori* and the argument validation code will be made part of the window system.

By separating the configurable portion of the interface, we permit the user to configure his interaction style easily—via the user configuration file (UCF), another component of UNCLE. In the remainder of the section we will look at the structure of a PCF, the encoding scheme and the mechanics of parsing the PCF. In the next section we look at the UCF closely.

3.1 What is a PCF?

A PCF is a collection of specifications—one for each command in the program. Each specification is divided into five parts: the syntactic component, validation component, application interface component, error message component, and documentation component. Each component is separated by a different separator as an aid towards parsing the PCF. Within each component and each specification, there is no constraint on the number of lines, just as there is no limit on the number of specifications. Any component can be safely omitted—presence of any component is purely optional.In the next section we will see how each component is encoded.

3.2 PCF encoding scheme

A syntactic description of the PCF is given in Figure 1. The first part of a PCF specification is the *syntactic* component. The syntactic component consists of the name of the command, a corresponding generic command (if any), followed by flags, options, and arguments on separate lines. Also, it has the name of the actual function that will be invoked to handle the command along with a one line message explaining the command.

A flag has the keyword "flag" followed by the string representing the flag and a brief description of the flag. The oneline description is used when the user requests information on the flag. The option line is similar: a keyword "option", followed by the name of the option and a brief description. A command can have an arbitrary number of flags and options. If *none* are encoded, it is assumed that the command has no flags or options. Each flag and option is encoded on a separate line.

Following the options, the arguments of the command are listed one per line, together with their types. Similar to flags and options, there is no upper or lower limit on the number of arguments that can be specified in the package configuration file. Once the flags, options, and arguments have

```
specs:              specs  spec
        |           spec
spec:               syntax_comp code_comp err_comp doc_comp
EOSPEC
syntax_comp:        GenericCommand Command flag_opt_args
flag_opt_args:      flaglist optionlist arglist
flaglist:           flaglist flag
        |           flag
flag:               FLAG COLON Flag_name Flag_Desc NEWLINE
optionlist:         optionlist option
        |           option
option:             OPTION COLON Opt_name Opt_value Opt_Desc
NEWLINE
arglist:            arglist arg
        |           arg
arg:                ARG COLON argument_names NEWLINE
code_comp:          validation_comp appl_intfc_comp
validation_comp: <code>
appl_intfc_comp: <code>
err_comp:           err_msgs
err_msgs:           err_msgs err_msg
err_msg:            SYMBOLICERR COLON error_message_string
doc_comp:           documentation
```

Figure 1: PCF encoding scheme

been specified, a line consisting of the name of the actual function to be invoked along with a one-line documentation returnable as a quick help for the command, is encoded. A separator signifies the end of the syntactic component.

The second component is the *validation* component. This component is responsible for ensuring that the user-specified flags and options are correct. Symbolic error codes are returned when errors are detected. If the validation code is omitted, the validation of flags/options is automatically performed by a generic routine. The validation code in the PCF is specified only in special cases to override the generic validation. The name of the validation code routine, if present, is recorded to be used when the matching command is issued.

The third component is the *application interface code* component. The application interface code enables the interface programmer to provide an extended channel of communication between the user and the application program. In most cases, the interaction between the user and the application program is a single command line. However, in certain cases, a command may require further input from the user. Such interactive gathering of secondary input is required only for a few commands and an alternative input gathering mechanism is needed. Secondary input gathering is application program specific and thus non-generic. For example, if the user would like to compose an electronic mail message, he would issue the *compose* command either via the keyboard or by selecting it via the mouse from the command menu. After the command has been received, the mail application program may decide to display a window with a template for a message and wait for the user to fill in the various fields of the template such as addressee, subject, and the body of the message. Once the user completes the composition of the message, another command may be required to *deliver* the mail.

Supplemental input gathering of this nature, while not as common as the simple single command line interface, is common enough to warrant interest. As our goal is to address the totality of the interaction process, we provide the application interface component as a means to address the supplemental input gathering problem. Further, we have insisted on the user having maximal control over the process of interaction. If we left the supplemental input gathering in the hands of the application programmer, we are faced with the same problem as before—a lack of uniformity and

customizability. The name of the application interface code routine, if present, is recorded for later use, similar to the validation code routine.

The fourth component in the PCF is the *error message component*, consisting of error messages returned at the time of parsing the user's commands. The error messages are gathered in the PCF for convenience. The line consists of the symbolic error code followed by the description of the error message.

The last component is the documentation for the command, which can be of arbitrary length. The documentation is saved to be displayed when the user requests documentation via the generic help facility. As this component ends a specification it is followed by the specification separator.

Any of the components can be omitted. The PCF parser does not require that the interface programmer who constructs PCFs to have validation or application interface code for all the commands. It instead uses dummy functions if one is not provided.

The PCF is thus made of two logically distinct components. The information regarding commands, as well as validation code, error messages, and documentation is independent of the user's interaction style. The application interface component, however, is oriented towards the needs of the particular commands for which supplemental input is required or for which special input gathering techniques need to be applied. Further, the application interface component is likely to have window system specific information in it unlike the rest of the PCF. This component may thus have to be changed when the PCF is ported to other window systems. The section on UCF will examine how the combination of PCFs with UCFs provide the desired flexibility in all the input gathering.

As an example, we present a piece of a PCF for the *RCS* application program. (Figure 2). The validation routine is in C.

The validation code in the example merely parses the flags and options returning an error message if warranted. If the interface programmer decides that only such simple validation is to be performed, he can omit it completely as the window system will do this by default. For the sake of completeness, we presented the code here. Thus the specification for the *show* command has the name of the command and the arguments listed in that order. After the validation code, the symbolic error message EARGTYPE is specified with the corresponding message to be displayed

```
"" ident
flag: q "suppresses warning if no patterns in a file"
arg: file
RCSident "identify files"
% validation_code
Validatercsident (csb, rgsb)
int csb;
char **rgsb;
{
    int i, irgsb = 1;

    if ((strncmp (rgsb[irgsb], "-",1)))
        if (!(strncmp (rgsb[irgsb], "-q",2)))
            return (EIDENTUSAGE);
    return (-1);
}
% error_message
EIDENTUSAGE: Usage: ident -q [file ...]
% documentation
Ident searches the named files, or, if no file name appears,
the standard input for all occurrences of the pattern
$keyword:...$ where keyword is one of author, date, header,
id, locker, log, revision, RCSfile, source, or state.
@
```

Figure 2: Sample PCF fragment

if the error is triggered. This is followed by a brief documentation on the command and its usage.

Thus, the PCF gathers the syntactic information and lets the window system provide completion, different styles of interaction, error messages in error windows and so on. In the next section we see how a PCF is parsed.

3.3 Parsing a PCF

To automate the process of recognizing PCFs for a wide variety of application programs, we constructed a parser using the UNIX tool *yacc*. Our encoding scheme enabled the writing of a simple parser that could extract the necessary information from the package configuration files. As the parser is written using a high level tool, it can be easily changed to accommodate modifications in the structure of the PCF.

The PCF parser extracts the various components from the PCF. First, the commands are read in and associated with the generic commands. The generic command specified in each of the PCF specification is bound to the command that follows it. We defined mode as a context-sensitive state and the current mode is named after the package. Thus the mode of a RCS command is "rcs". A table of the commands are formed for a menu-style interface. The command names are also linked to the actual function that has to be invoked.

3.4 Construction of the interactive system

The information gleaned from the PCFs is encoded into header files and included in the validation code file generated. The validation code file is compiled into the window manager. The window manager makes no assumption about the number of the PCFS or their names. The window manager has a dummy include file into which information specified at interactive system generation time is inserted.

To build the interactive system, the interface programmer constructs the various PCFs and issues a single command with the names of the PCFs. The command parses the PCFs using the generic parser, downloads the generated code into the window manager, compiles the window manager, and begins executing it. The list of application programs for which PCFs were specified are available as a menu in the root icon. The user selects

the application program via this menu and starts issuing application program specific commands. The various interaction styles provided and their construction techniques are discussed in section 4.

The validation code ensures that the flags, options, and arguments are correct. If a flag is not recognized, the list of valid flags parsed from the first component, together with the associated one line help message is display in an error window.

3.5 Further thoughts on PCFs

Once the PCFs are constructed, the interface can be presented in different ways. For example, the commands can be presented in a menu. As we have already separated the parsing from the intended action on the parsed items, we can present different interaction styles without altering the PCF.

Unfortunately, as the PCF is a static entity, and as the entire interactive system is configured from the PCFs, it is impossible to change the bindings that are being made at this stage. To permit dynamic alterations, we need to read in a different configuration file. The implementation is currently in C, a language that does not permit late binding, making it infeasible to provide dynamic alteration. Implementation in a late-binding language like LISP or Smalltalk would solve this problem.

Keeping in mind that the potential audience intending to build PCFs are interface programmers, we could conceivably move to a more concise format. Another option is to dynamically gather the necessary PCF information, whereby, changes in the application program would be transparent to the window system as the information would be gleaned from the application program at invocation time.

4 User configuration file—UCF

PCFs represent the break of the link between the application programmer and the user. While PCFs relieve the application programmer of the responsibility of providing a flexible interface, the UCF (user configuration file) represents the scope of control of the user over the interface. Our main goal is to reduce the workload as well as the hold of the application programmer over the interaction process and maximize the flexibility in the

interface for the user.

We begin by examining the different aspects of interaction. The user cannot have any control over the actual execution of commands as the semantics of commands are solely under the aegis of the application programmer. However, the user must have a free hand in the method of input gathering, the actual physical input events to be issued to invoke commands, the choice of a interaction style, the physical attributes regarding output location, and display style. At the same time, he should not be forced to specify any defaults—the window system should have defaults for all aspects of interaction. The user should be able to specify via a configuration file and be able to dynamically alter any of the interface features. If interaction decisions are not bound at a low unchangeable level, the user can change his mind at any stage. By being able to alter the interface features dynamically for a specific application program, the user can try out a different style for that application program.

4.1 Structure and components of a UCF

In the UCF, we permit the user to specify global and local attributes that control his interaction process. Global attributes differ from local attributes only in the sense that they apply to *all* the application programs. The user can choose global defaults and thus avoid specifying them for each application program. The various attributes he can specify include:

- choice of location for default window positions
- global default style of interaction
- choice of physical input events to be used as:
  - — completion event
  - — event to list set of valid choices
  - — event that provides context-specific help.

The choices can be overridden for one or more application programs by either specifying the particular choice for the application program in the UCF itself or by changing it while the application program is running.

The user specifies the choice of an interface feature along with its value by prefixing the attribute with either the keyword 'global' or the name of the application program. This style of specification follows the style popular with programs written to run under the X window system. X, in turn, had adopted the style of the configuration file belonging to the Andrew window system [2].

For example the specification

```
global.style: buttonstyle
```

implies that the user would like to interact with all the application programs in the *button-style*, whereby the set of valid commands of a particular application program are displayed as buttons on the corresponding window. Similarly the specification

```
global.Help-Event: '@'
```

implies that the input event '@' is bound to the invocation of generic help. The actual routine invoked corresponds to the current status of the command line being constructed. If the user has constructed the name of a command and then issues the Help-Event, documentation on the command would be displayed. If he has constructed part of the command and issues the Help-Event while specifying an option or a flag, help on that flag or option is displayed.

Now, suppose the user wishes to specify a different physical input event as the help event for a particular application program. He merely adds another specification in the UCF as follows:

```
fooAP.Help-Event: 'X'
```

In the application program *fooAP* alone, 'X' would be the help event.

The idea is not novel: the notion of a particular binding in an application program is akin to that of local bindings in Emacs. What is different however, is that the variables are not bindings to editing commands but to more general, logical notions such as completion, help etc. These bindings are almost always bound at a low level in practically all interactive systems and it is our intention to show that it can be bound at a much higher level *and* changed at will, both locally as well as globally.

4.2 Interaction between the user and UNCLE

There are four levels of setting the values of the various interface features:

- Global defaults of the window manager.
- Global defaults specified by the user.
- Application program specific defaults specified by the user.
- Local and global values set dynamically during a session.

The levels are in increasing order of importance and the higher level specifications override the previously specified values. We consider these different levels essential for providing complete generality to the notion of customizability. Absence of any of the levels signifies an artificially introduced constraint.

An important facet of our model of an interactive system is uniformity, exemplified by the user's capabilities in dealing with interface attributes. Defaults are provided at all levels. The defaults can be overridden statically via the UCF or dynamically by interacting with the window manager. Completion is provided at all stages. While constructing the command, the list of valid commands can be obtained by issuing the *list-completion-event*. The same event can be issued to obtain a list of the valid flags and options for the command if the command has already been typed.

Once the user has created an application program window, he can only issue commands that are specific to that application program. Depending on his choice of style the user is presented with a fixed-menu of the set of valid commands, a pop-up-menu, or just a keyboard interface (section 5.2 discusses the various input styles provided in UNCLE). All these styles were generated from the same PCF and the application programs were not modified. Our contribution to interface design is felt here—a style of interaction chosen by the user at no extra cost to the application programmer and very little cost to the interface programmer. The user selects a style that he is comfortable with but can change his style on the fly. He can interact with *all* the application programs in the same style or in different styles. The choice of style of interaction should be of no concern to the application programmer whose task is the construction of the application program alone. Even after choosing a particular style, the user has plenty

of flexibility in interacting with the application program. For example, he can, from within an application program window alter the completion event for that application program. He can change any of the interface details such as input and output style, bindings to generic events etc. Further, the interactions in all the styles are uniform. We have already mentioned how a user can obtain documentation of commands, flags, options at any stage of the command line construction by using the generic help facility. Error messages are displayed in a separate window so as not to clutter the interaction window. The documentation window and error window can be unmapped by simply clicking in those windows. The bindings to invoke documentation, or the command are the same regardless of the style (pop-up or fixed menu).

The role of the UCF extends beyond input—the user can specify his output style as well via the UCF. For example, he can specify *a priori* that the output for a particular application program should be displayed in a separate output window. The physical attributes of the output window, such as location, size, and display font can also be specified in the UCF. Just like any other attribute, output location has a default—the same window in which input is gathered. Output location can be dynamically altered while interacting with a particular application program. The user can request future output to be displayed in a separate output window. He can also request that output for a particular command should be displayed in its *own* window, which can then be used for subsequent modification of the output. The interactive system does not attempt to impose any restriction on where and in what fashion the output should be displayed. The role of the interactive system is that of an agent assisting the user. Consequently, it does not prevent the user from choosing and altering both input and output styles.

The UCF also plays a role in the supplemental input gathering mechanism implemented via the application interface component of the PCF. The application interface component interacts directly with the UCF when the application program is running, by inquiring how the user would like to interact with a specific command that requires further user input. We will consider the composition of mail example again. To create a new window for constructing the message, and having it deleted automatically upon message delivery, the user simply needs to alter his UCF. By specifying his choice of options and changing them dynamically, he can control the

interface to the *compose* command.

All the interaction details have thus been gathered in a single place. We then extended our model to permit global alteration of the interface model of a user. We first permitted local modifications of interface features present in the UCF. Next, we permitted the user to edit his configuration file, make multiple modifications and issue a command to have the modifications reflected in his environment. A corollary was the ability to read in *another* user's UCF. Before reading in the UCF belonging to another user, we save the present environment, to enable the original user to switch back to his environment.

Thus, the UCF is the vehicle with which the user exercises the various choices provided by the window manager. The window manager in turn derives its power from the PCF. The PCF, together with the UCF contributes to a uniform, customizable, and configurable interface.

4.3 Extensions to the UCF

The UCF was born out of the perceived lack of uniformity in existing interactive systems and the needless low level binding of physical input events to logical functions. For example, it is hard to find an interactive system where a logical function such as completion is not bound to a predefined, unchangeable physical event. In its present state, the UCF leaves room for expansion. A small extension would be the ability to specify different mouse style interactions. A common problem in several interactive systems is the direct binding of mouse events. As we do not view mouse input to be different from keyboard input, we do not think physical mouse input events should be hardwired any more than keyboard events. Mouse events should be treated completely analogous to keyboard events and thus logical mouse action should be bound to functions. An example of a logical action done via a mouse is *selection.*

5 UNCLE implementation under X

After identifying the components of our model interactive system to be the PCF, UCF, and a sample set of application programs we set about the task of implementing the model interactive system on top of an existing

window system. As we felt that our ideas were fundamental in nature and not dependent on any particular window system we chose a window system that could simplify our task. We merely require the following of a window system:

- Ability to create identifiable rectangular regions on the screen.
- Ability to interpret mouse as well as keyboard events.

Beyond this, we did not require anything of the window system. We did not want to worry about the idiosyncracies of the window system that we chose. For example, the VGTS window system of the V-kernel does not relinquish control over the mouse easily. Mouse events had to be interpreted in a particular manner. We needed the freedom to rebind the mouse events as well as keyboard events and preferred a model that merely obtained input events from input devices allowing the programmer to deal with them freely. As the author was quite familiar with the C programming, a window system that could be programmed under C was preferred. Though, NeWS (Network extensible Window System) had the notion of extensibility built into the design of the window system, the lack of its availability as well as the choice of PostScript as a programming language deterred the author from adopting it as a vehicle for implementation. After completing the implementation, we have reasonable grounds to state that our model can be implemented under other window systems just as well. The choice of X permitted the usage of user interface packages such as the XMenu package, that made our prototype construction easier.

In the rest of the section we discuss the implementation by looking at the information extracted from the PCFs. This is followed by a look at the various input and output styles implemented and a discussion on interaction with the window system itself. The execution model and the ability to modify the entire environment is then discussed. We conclude with a look at the help facility in UNCLE.

5.1 Information generated from the PCF

Barring the application interface component of the PCF, the rest of the PCF and the whole of the UCF are window system independent. Once the PCFs are written, the generic package configuration file parser (PPCF)

would parse all the PCFs specified and generate three files for each PCF (Figure 3).

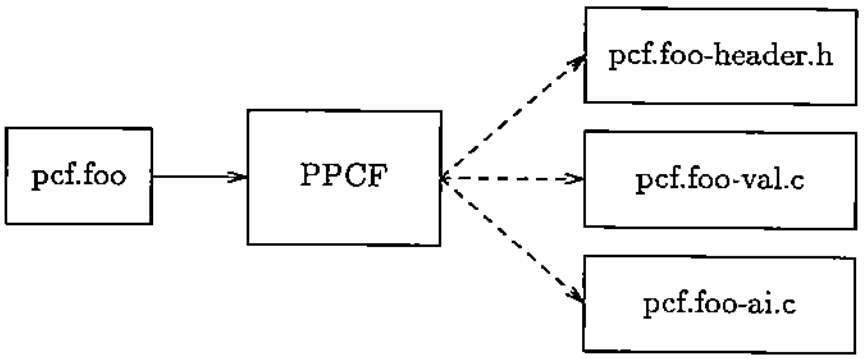

Figure 3: Parsing a PCF

The first is the package header file that includes all the declarations for the validation functions, binding of the symbolic error messages to integers, and various structures that are used when the application program is run. The structures include

- a map from the error message numbers to corresponding strings
- a map from the command to the validation function to be invoked
- a structure that includes:

  the command

  matching generic command

  flags, options, arguments with brief documentation

- documentation for the command.

The information extracted from the PCF was converted into C structures and compiled into the window system.

The second file generated is the validation code file. The validation code file has routines that would be called for each command whose flags, options, and arguments needed to be validated. The PCF parser would extract the names of the functions and declare them in a header file so that they could be part of a structure that mapped the names of the commands

to the corresponding validation function that had to be invoked. If the interface programmer had felt that a particular command did not need to have any validation performed on it and had omitted validation code for that command, the parser would automatically use a default validation function in its place.

The third file generated is the application interface code file, also extracted directly from the PCF. This file consists of special routines that are called *after* the initial parsing of the command line. As explained in the section on PCFs, the application interface code provides the application program with a mechanism to gather secondary input. The alternate mechanism can be viewed as an escape hatch provided for certain special commands. To ensure that the user has maximum control over the process of interaction, we made the alternative (or supplemental) input gathering mechanism a part of the PCF that could be controlled by the UCF.

The three files generated are made part of the window system and the routined defined in these files are invoked at run time.

5.2 Input styles

To demonstrate the idea that different styles of interaction can be built based on our model for the same application program without modifying the application programs themselves, we constructed a few styles. Some of the styles were commonly available ones, e.g. a *shell-style*, whereby a user types a command line and the command is then executed. Other styles included:

- *Menu-style*: a pop-up-menu of valid commands displayed on request.
- *Button-style*: a fixed menu displayed as buttons in the window.
- *Form-style*: the command with flags, options, and arguments are displayed in a form and the user could fill in the items of his choice.

The major difference in our implementation of these styles is the uniformity that our model demands and the customizability it provides. In all the styles, the user can bind any physical events to logical commands such as *help*, *completion*, *list-completion*, and choose his style of output—window, different window for all output, different window for a specific command.

In other words, while the user chose a particular style of interaction, he could continue to enjoy a uniform customizable interface. In the reminder of this section we will briefly describe the various styles of interaction constructed for the four application programs. Some of the styles of interaction were more suitable for a particular application program than other. The user could specify his choice of a style of interaction on a per-application program basis and change it at any time.

5.2.1 Shell-style

Shell-style is a popular style of interaction. On ordinary terminals it is the only style available. The user constructs a command line by typing in the command, flags, options, arguments and the window system after verifying the command line sends the command off to be executed and returns the output. By default the output is displayed in the same window where the user issued the command.

The UNCLE window system provides the user with command completion for the list of valid commands in the current application program. Help is available at every step of the command line construction. By issuing the keystroke bound to the *list-completion* function, the user could get a list of valid commands in the present application program. If the *list-completion* function is invoked after typing the name of the command, the list of valid flags and options are displayed. Two other special input events exist: one that specifies completion of the command line and another that dynamically alters the output location of the current command alone. The former (usually the *return* key) specifies that the command has been completed and should be executed. The latter is the *ownwin* event and output of the command is displayed in its *own* window. The *ownwin* event is discussed in more detail in the section on output styles (Section 5.3). If the *generic-help* event is issued after typing in the name of the command, documentation on the command is displayed and if it is issued after specifying an option or a flag, explanation of the flag or option is displayed. UNCLE, thus keeps track of the current state of the user and provides context-specific help.

5.2.2 Menu-style

In the menu-style of interaction, a pop-up-menu consisting of the valid application program specific commands as well as the window system commands are displayed (in separate panes). The menu is invoked by issuing a mouse input event and the user can either execute a command by selecting it or avoid making any selection by moving the mouse out of the menu area. Also, this a way to display the list of valid commands in the application program. As the entire command is displayed no completion is necessary. However, if the command has any flags or options or requires any arguments, the user is prompted for the same via a form. If the command has no flags or options, it is simply executed as soon as it is selected from the menu. The last menu item selected is the default selection for the next invocation of the menu. The advantage of the menu-style is that the user can see the commands and select the appropriate one. Only valid commands are displayed and thus command specification errors are eliminated.

5.2.3 Button-style

In the button-style of interaction, the valid commands are already displayed as buttons. Thus, this style can be viewed as a single fixed menu. The user moves the mouse to the button representing the command he wishes to invoke and clicks a mouse button. The differences between the button-style and the pop-up menu-style are obvious: in the former, the menu is already visible but the user needs to move his mouse to the buttons, while in the latter the menu can be displayed anywhere. The drawback in the menu-style is that the menu has to be explicitly displayed each time a command is to be issued.

We have made our model uniform enough that the generic events, such as *help*, can be issued in the button-style as well. The user moves the mouse to the button and can invoke help on the command, or execute the command in its own window by issuing the ownwin input event, just as he would do from any other style. Generic events are independent of the current style of interaction as well as the current context.

5.2.4 Form-style

The form-style of interaction is mainly for users who need more guidance in constructing the command line. When the user issues the command, a form is popped on the screen nearby. The form has the command name in it and depending on whether the particular command has flags, options, or arguments, a corresponding entry is present in the form. If a command does not have flags, options, or arguments, no form is popped up. After the form has been displayed, the user can skip over any item without having to specify a value if it is optional. If the user does type in a value for the flag or option item, it is immediately checked for validity by the window system. The user can obtain a list of valid flags or options by issuing the generic *list-completion* input event at this stage. After typing in a flag or option name, he can obtain documentation on it by issuing the generic-help input event. When the complete command has been constructed, the form will be unmapped and the command executed.

The form-style can be turned on and off independent of the other major styles of interaction. The user can use the form-style while in shell-style. In this sense, the form-style is not a distinct style of interaction; it is a sub-style of the other styles. The novice user can make use of this feature while the advanced user can turn this feature off.

5.3 Output styles

One of the main thrusts of our model of interaction was to separate input and output from the application program, enabling the user to control it. In the previous section, we showed how the user can use different input styles interchangeably. In this section, we will show how the user can also control his output style. The advantages of letting the user decide how he wants the output to be displayed, apart from the obvious added customizability, is that the application programmer's task is further reduced. We have already moved the input gathering task from the application program to the window system. By obviating the application programmer from having to deal with output display, we have further streamlined the application program. The task of the application program is now simply executing a set of commands with no interface details buried in it. Maximum control is placed in the hands of the interface programmer, who in turn enables the

user to exercise any degree of control.

When a command has been validated for syntactic correctness and sent off for execution, the returned output should be under the control of the user to display it in any manner. By default, in most interactive systems the output is displayed in the same place where the user has typed in the input. Similarly, in UNCLE, output by default appears in the same window unless the user specified that output should always be redirected to a separate output window. Yet another possibility is displaying the output of a command in a window by *itself*. If after constructing the command line, the user wishes to redirect the output of the present command to a window of its own he can do so by issuing the *ownwin* event. The *ownwin* event (just like any logical event, this can be bound to any physical input event of the user's choice and changed at will) tells UNCLE to display the output of the command in a new window. With output displayed in its own window, *filters* can be used on the data. For example, the user can *search* for a string in the output or *sort* the output. The sorting can be done *in-place*, i.e. in the same output window, or the sorted information can be displayed in yet another window.

As with all the interface features in our model, the user can state *a priori* what output style he prefers. He can specify a global output style if he wants to, and selectively override it for a particular application program. After entering UNCLE, he can change it locally for a single application program or change it globally. He can have the output of a particular command sent to its own dedicated window or have the output of all commands redirected to the same or new windows. A combination of these output styles is even more powerful as the user can decide if he wants the output of a *particular* command to be displayed in a window by itself.

5.4 Interaction with UNCLE

In the preceding two sections we have looked at the various input and output styles with which the user can interact with the various application programs. We now consider interaction with the window system itself. According to our model, the interface between the user and the window system should not be different from the interface between the user and the application programs. We treated UNCLE to be yet another application program and constructed a package configuration file for it. The valid

commands in this application program are all the possible interactions with the window system. The list included commands to change input and output styles of interaction, and generic input events. The PCF for UNCLE is similar to any other PCF in structure and the generic parser was used to parse this PCF too. Thus, *uncle* would be one of the application programs that can be invoked from the UNCLE icon. He could run the UNCLE application program and change his styles of interaction and generic input events globally.

For the intra-application program changes of interface features, we appended the list of valid UNCLE commands to each of the application program's PCF. The addendum contained the commands that the user could invoke to interact with the window system.

5.5 Model of execution

The model of execution is as follows: an icon representing the interactive system (labeled UNCLE) is displayed after all the PCFs have been parsed and the window manager compiled. The user selects application programs via a menu in the UNCLE icon. When an application program is selected, UNCLE forks off a process with the application program running in a new window. The window is created in the style specified in the user's UCF. The user's specifications in the UCF can include physical location details about the application program window. As and when the user creates application programs, the parent (UNCLE) process kept track of the process ids of the child processes created.

To alter any of the interface attributes *globally* the user would interact with the *uncle* application program (section 5.4). For example, if the user decided to change the completion event globally, he would be prompted (in a pop-up-window) to type in his new choice. The uncle application program would deposit a line of the form

```
Completion-Event: value
```

in a predefined file and signal the parent UNCLE process which in turn would signal all the application programs. The file system was thus used as a rendezvous point. As the version of the UNIX operating system being used by the author on the Sun workstations lacked shared memory capabilities, the signal facility of UNIX was used for inter-process communication.

The individual processes running the application programs had signal handlers to handle asynchronous communication from the parent UNCLE process. Each of the child processes' signal handler would read the rendezvous file and alter the local completion event variable accordingly.

If however, the user wanted to change interface attributes for a particular application program alone, he would invoke the options menu of the application program and select the attribute that he wishes to alter. The code to gather input from the user to change local or global attributes is the same, adding to the uniformity of interface.

As mentioned earlier, the user can dynamically alter the location of output display between the same window, a separate output window, or to a command specific window. The ability to specify and alter the output location via the PCF and UCF shows that the application programmer tasks in the area of displaying output can be reduced. We believe that further research in the area of input gathering from non-traditional sources (e.g. discontinuous portions of, or an entire window), redirecting output from different sources to a single output window etc., is necessary.

In the following subsection we will see how the environment can be totally altered dynamically to accommodate a different user. Also, we discuss how help is provided and how a history mechanism keeps track of the user's commands under UNCLE.

5.6 Changing user environments

We have seen how the user can specify and dynamically alter his bindings to the logical commands, as well as his input and output styles. The totality of a user's *interaction environment* can be specified in the UCF and modified via UNCLE.

When a user wants to briefly use some one else's workstation, he is faced with the problem of a potentially alien interface. If it were possible for him to *temporarily* push his environment on to the workstation he would be on familiar terrain. The alternative is restarting the interactive system with his environment, which is time consuming and disruptive to the current session.

We provide a mechanism by which UNCLE can read in the UCF of another user. The interface to the existing set of application programs would be changed to reflect the style of the new user. For example, if the

choice of location and size of the windows corresponding to the application program are different, they are changed accordingly. The bindings to the generic events, the input style (button-style, menu-style etc.) as well as the output style (same/new window) are all set according to the new user. However, before making any changes, the present state is saved to enable later restoration. Thus, the new user could interact with the various application programs and the window system as if he had originated the interactive session at that workstation. Later, the original user can invoke a *restore* routine to go back to his configuration. A trivial extension is to provide a multi-level backing rather than a single save and restore, whereby a series of users can push and pop their environments onto a single interactive session.

The routine that permits pushing another user's environment, by default permits the user to modify *his* configuration file and re-initialize his environment reflecting the changes. The user can thus experiment with different styles for one or more application programs. As each application program is running under the control of a separate process, pushing the environment of another user from a particular application program would only affect the interaction with that application program. However, by interacting with the UNCLE application program, the user can modify his interaction globally. The same environment modification routine, when invoked from the UNCLE application program, notifies all the application programs to read in the new configuration. Similarly, the restoration routine when invoked from the UNCLE application program, will restore the complete environment of the user.

The environment modification ability of UNCLE permits the user to modify the interactive system dynamically. He can thus take a *snapshot* of the session and revert to the snapshot state after making further changes. The snapshot not only gathers the physical sizes and locations of the windows in the interactive sessions, but all the interaction details such as bindings and input/output styles.

5.7 Help

Documentation on the application programs, its commands, flags, options, and arguments are useful to novice users and to a lesser extent to expert users. Our primary criterion is to provide documentation uniformly, irre-

spective of the context and the user's current style of interaction. Help is available at every stage of command construction. The user can find the list of valid commands before commencing the command line construction, and obtain documentation on a particular command after typing it in. He can do the same for flags and options of the command after specifying the command. Obtaining help is independent of the style of interaction. For example, in button-style, the user can obtain help on a command by issuing the generic-help input event while the mouse is on the button. Similarly, in the form-style of interaction, the user can issue the generic-help input event while filling in the items of the form. Also, help for the window system commands can be obtained in a manner similar to that of application program commands.

6 Conclusion

We discussed the implementation of a window system based on abstraction mechanisms by virtue of which we are capable of providing a uniform as well as customizable interface in a variety of interaction styles. We suggested the use of logical input events removing low level bindings of input events to functions in application programs. The user could select his bindings, change his styles of interaction on the fly and switch environments—all during the course of a session.

The advantages of the UNCLE approach was the reduction of work for the interface programmer in providing different interaction styles. A configuration file could be easily generated for each application programs and the generic parser (part of UNCLE) could be used to parse this configuration file. The user could have application program specific styles of interaction or a uniform style for all application programs.

The drawbacks of the present implementation are the static binding of configuration details. With a late binding language we could dynamically read in new configuration files. The UNCLE model cannot be applied to all application programs, for example, an editor. The inability to apply the UNCLE model to an editor like program stems from the fact that the granularity of interaction between the editor and the user is at the character level. Our model functions best for application programs with commands as communication units.

References

[1] D. Comer. *Operating System Design, Vol. 2: Internetworking with Xinu*, chapter 15: A Syntactic Namespace, pages 239–261. Prentice-Hall Inc., 1987.

[2] Information Technology Center. *Users' Manual of the Andrew System.* Information Technology Center, April 1985.

[3] B. Krishnamurthy. *Partitioning the Process of Interaction: An Abstract View.* Technical Report CSD TR 705, Department of Computer Sciences, Purdue University, September 1987.

[4] B. Krishnamurthy. *Shells In An Interactive System.* Technical Report CSD TR 707, Department of Computer Sciences, Purdue University, September 1987.

[5] B. Krishnamurthy and C. E. Wills. *Omicron: Events => Actions.* Technical Report CSD TR 584, Department of Computer Sciences, Purdue University, April 1986.

[6] M. T. Rose and J. L. Romine. *MH—Mail Handler.* The Rand Corporation, April 1986.

[7] W. F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering*, IEEE, September 1982.